IUT de Bordeaux
Département Informatique

M2201 – Graphes et Langages

Éléments de Théorie des Graphes

Introduction, définitions
Les graphes : un outil de modélisation
Représentation des graphes

Éric SOPENA, eric.sopena@labri.fr

Première partie

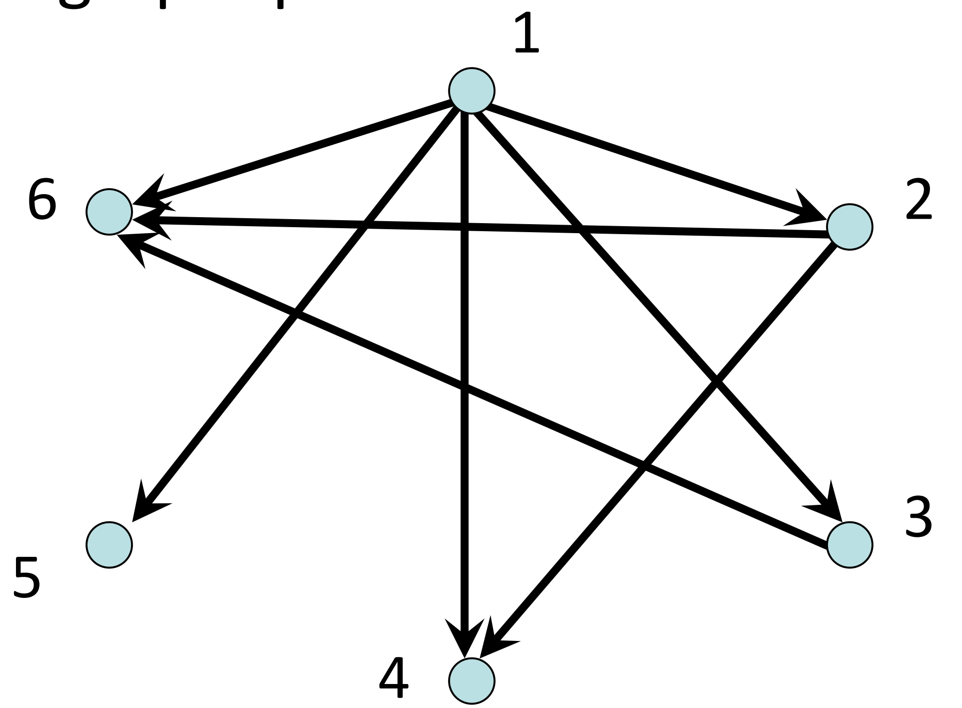
Introduction, définitions

Graphe d'une relation (petite école…)

Soit la relation R sur {1, 2, 3, 4, 5, 6} définie par :

x R y si et seulement si x | y (x divise y)

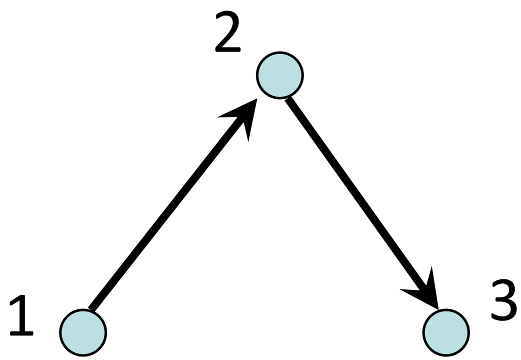
Représentation graphique :



Graphe orienté (définition)

Un **graphe orienté** G est un couple G = (S, A) où

- S est un ensemble fini de **sommets**
- A est un ensemble de couples de sommets appelés **arcs**



S = { 1, 2, 3 }

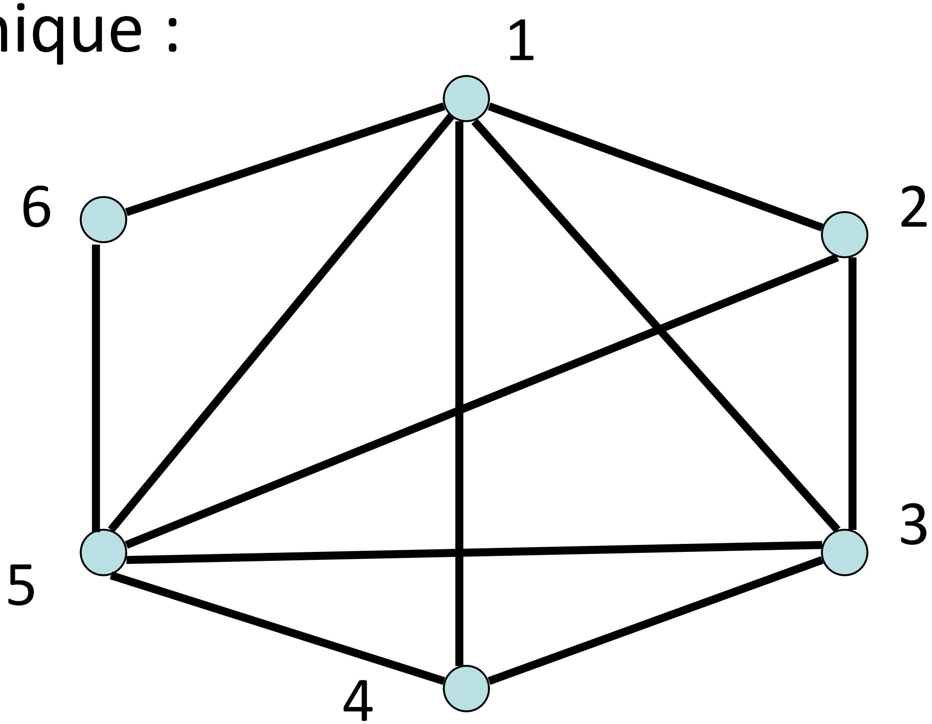
A = { (1,2), (2,3) }

Graphe d'une relation symétrique (1)

Soit la relation R sur {1, 2, 3, 4, 5, 6} définie par :

x R y ssi x et y sont premiers entre eux

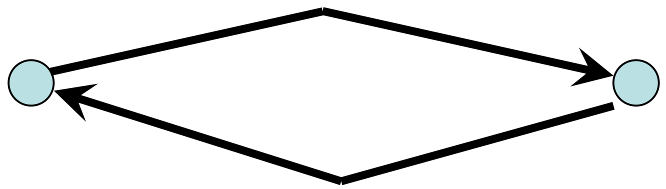
Représentation graphique :



Graphe d’une relation symétrique (2)

En réalité,

est un « raccourci » de dessin pour :



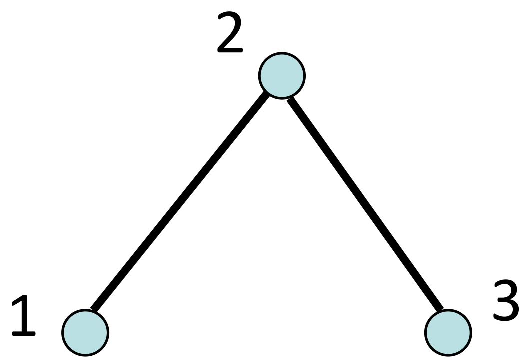
Nous avons un graphe orienté **symétrique**, appelé **graphe non orienté**…

Graphe non orienté (définition)

Un **graphe non orienté** G est un couple G = (S, A) où

- S est un ensemble fini de **sommets**

- A est un ensemble de « paires » de sommets appelées **arêtes**



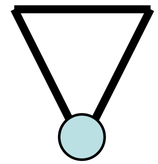
S = { 1, 2, 3 }

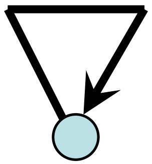
A = { {1,2}, {2,3} }

équivalent à A = { (1,2), (2,1), (2,3), (3,2) }

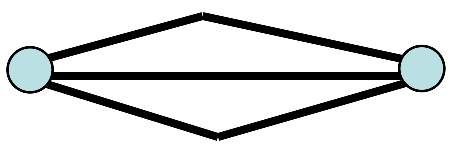
Boucles, graphes simples, multigraphes

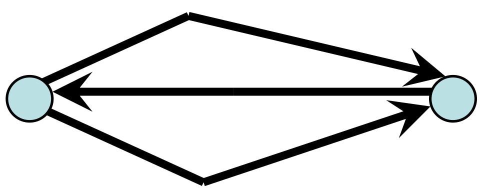
Une **boucle** :





On parle de **multigraphe** (orienté ou non orienté) si on autorise plusieurs arêtes (ou arcs) entre deux sommets (arêtes multiples).





On appelle **graphe simple** un graphe sans arêtes (ou arcs) multiples.

Degré d'un sommet, voisinages (1)

Cas non orienté

Si {u,v} est une arête, u et v sont **voisins**.

Le **degré** d'un sommet u, noté d(u), est le nombre de voisins de u (*attention : les boucles comptent double…*).

On note $\Delta(G)$ le *degré maximum* de G, et $\delta(G)$ le *degré minimum* de G.

Cas orienté

Si (u,v) est un arc, u est un **prédécesseur** de v et v est un **successeur** de u, u et v sont **voisins**.

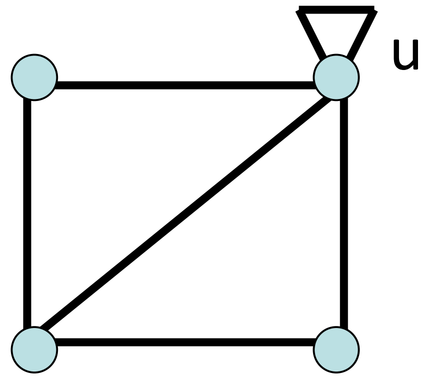
Le nombre de prédécesseurs de u est le **degré entrant** de u, noté $d^-(u)$, le nombre de successeurs de u est le **degré sortant** de u, noté $d^+(u)$, le nombre de voisins de u est le **degré** de u, noté d(u) :

$$d(u) = d^-(u) + d^+(u).$$

Degré d'un sommet, voisinages (2)

Exemples :

G :



$d(u) = 5$

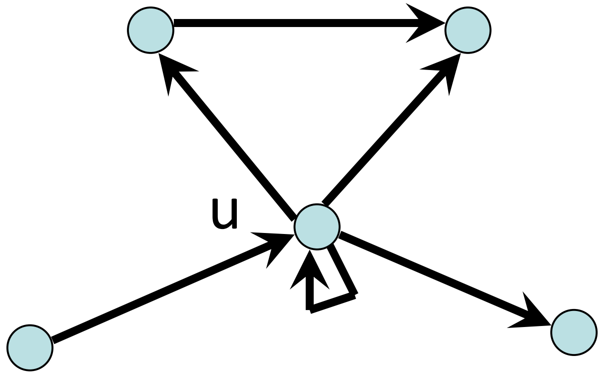
$\Delta(G) = 5$

$\delta(G) = 2$

$d^{-}(u) = 2$

$d^{+}(u) = 4$

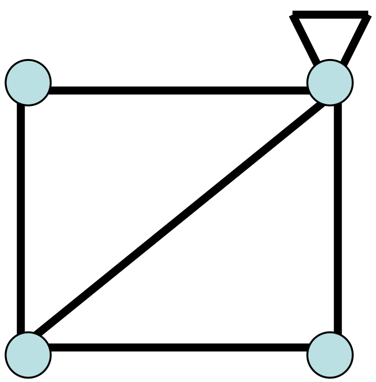
$d(u) = 6$



Degré d'un sommet, voisinages (3)

Pour tout graphe G, la somme des degrés des sommets de G est égale à deux fois le nombre d'arêtes de G (**lemme des poignées de mains**).

Pour tout graphe orienté G, la somme des degrés entrants des sommets de G est égale à la somme des degrés sortants des sommets de G et au nombre d'arcs de G (**lemme des coups de pieds**).



2 + 5 + 2 + 3 = 2 x 6 = 12

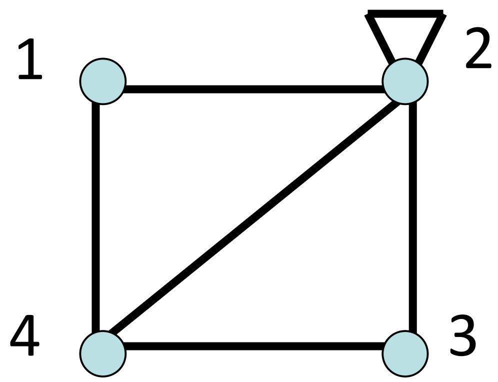
Récurrence sur nombre d'arcs ou d'arêtes…

Chaînes, chemins, cycles et circuits (1)

Une **chaîne de longueur k** dans un graphe non orienté est une suite de k+1 sommets $u_1, u_2, \ldots, u_{k+1}$, telle que pour tout i, $1 \leq i \leq k$, $u_i u_{i+1}$ est une arête (longueur = nombre d'arêtes).

Si tous les sommets sont distincts, la chaîne est **élémentaire**.

Si $u_1 = u_{k+1}$, la chaîne est un **cycle**.



12232412 est une chaîne

123 est une chaîne élémentaire

12243241 est un cycle

2342 est un cycle élémentaire

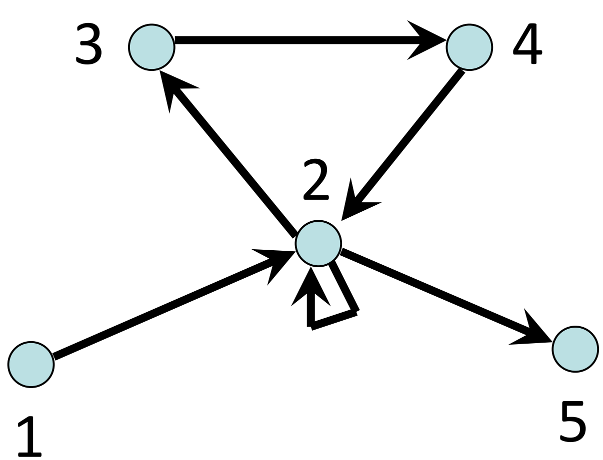
Chaînes, chemins, cycles et circuits (2)

Une **chaîne de longueur k** dans un graphe orienté est une suite de k+1 sommets $u_1, u_2, \ldots, u_{k+1}$, telle que pour tout i, $1 \leq i \leq k$, $u_i u_{i+1}$ ou $u_{i+1} u_i$ est un arc (longueur = nombre d'arcs).

Chemin de longueur k : $u_i u_{i+1}$ est un arc pour tout i.

Élémentaire : idem.

Circuit : chemin tel que $u_1 = u_{k+1}$.



322524 est une chaîne

1243 est une chaîne élémentaire

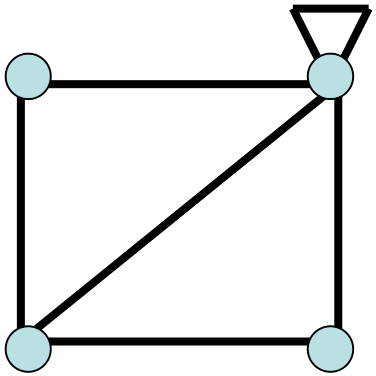
125 est un chemin élémentaire

34223 est un circuit

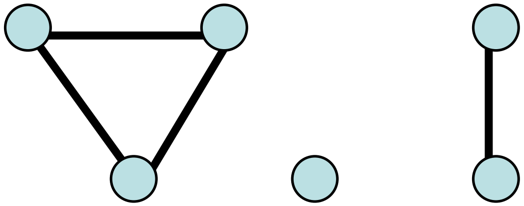
2342 est un circuit élémentaire

Connexité (1)

Un graphe non orienté est **connexe** si pour tous sommets u et v il existe une chaîne allant de u à v.



connexe

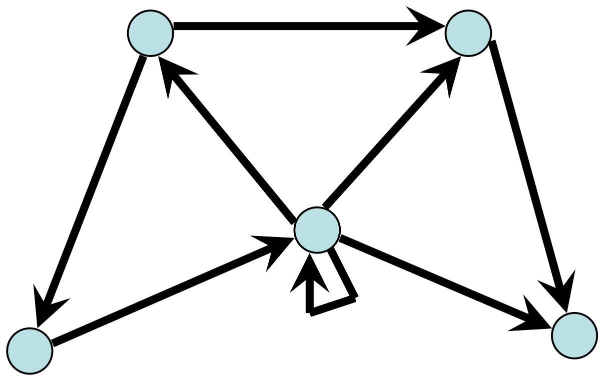


non connexe

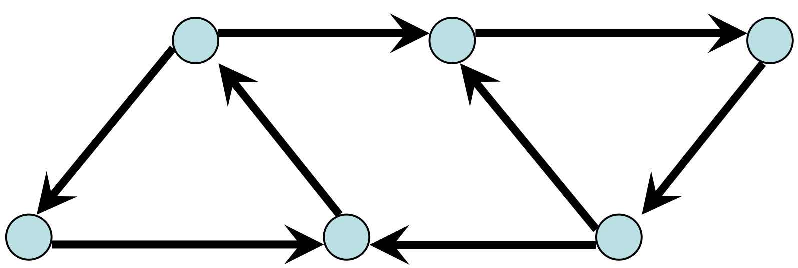
→ composantes connexes…

Connexité (2)

Un graphe orienté est **connexe** (resp. **fortement connexe**) si pour tous sommets u et v il existe une chaîne (resp. un chemin) allant de u à v.



connexe mais pas fortement connexe



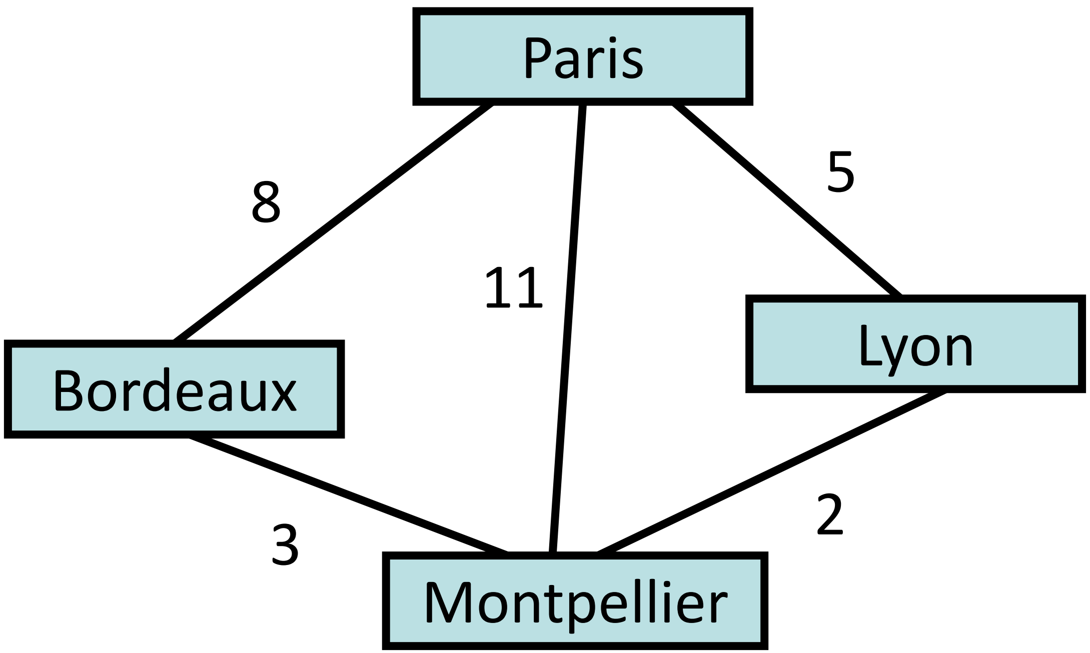
fortement connexe

→ composantes fortement connexes...

Graphes valués, graphes étiquetés

Un graphe est **valué** si on associe des valeurs à ses arcs (ou arêtes).

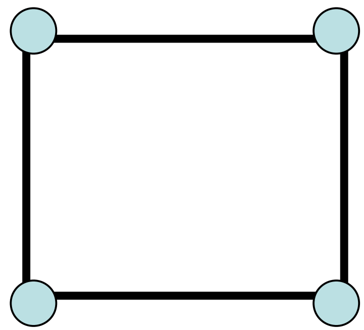
Un graphe est **étiqueté** si on associe des étiquettes à ses sommets.



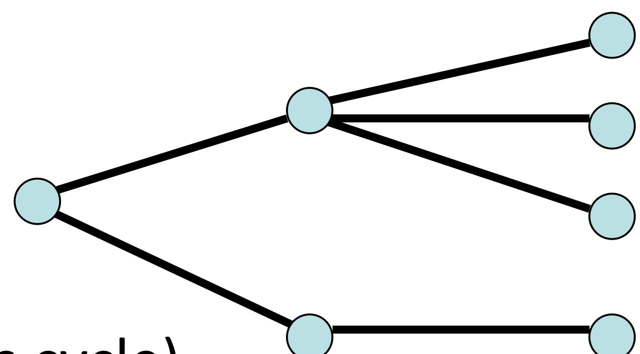
Types particuliers de graphes (1)

Une **chaîne** :

Un **cycle** :



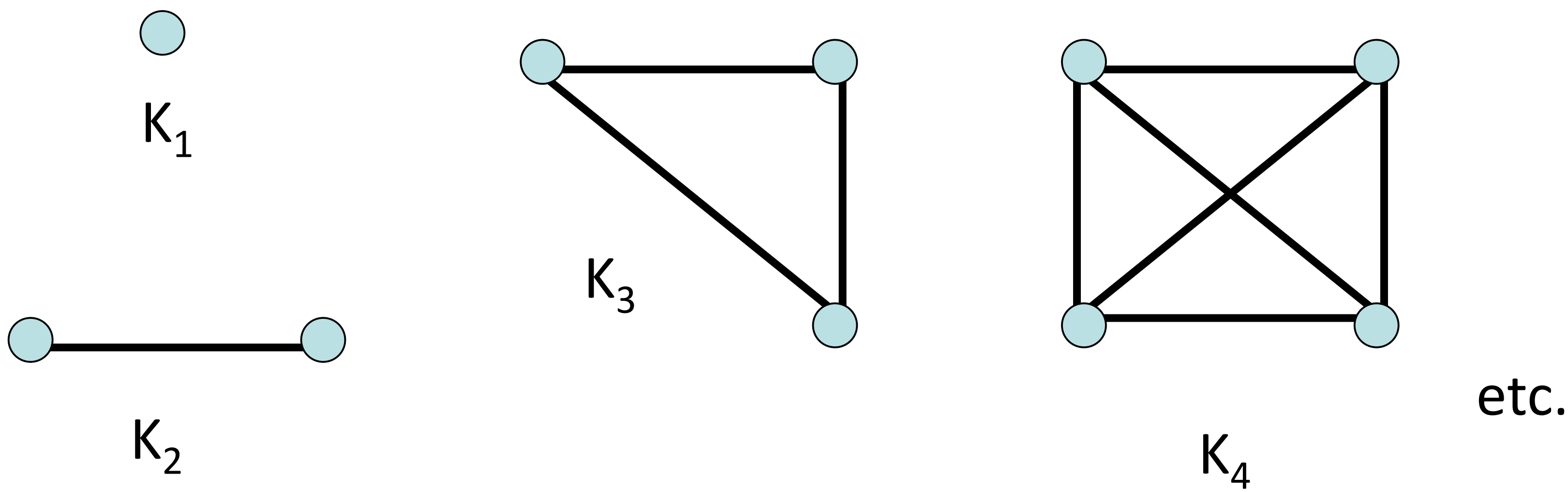
Un **arbre** :



(Graphe connexe sans cycle)

Types particuliers de graphes (2)

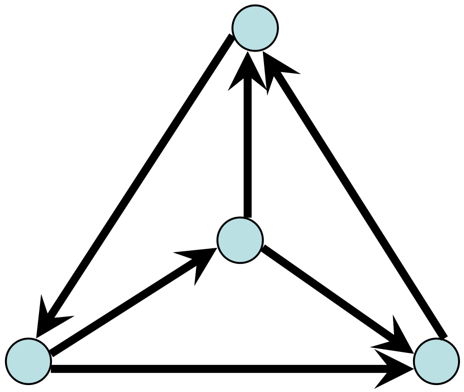
Un graphe est **complet** si tous ses sommets sont reliés deux à deux.

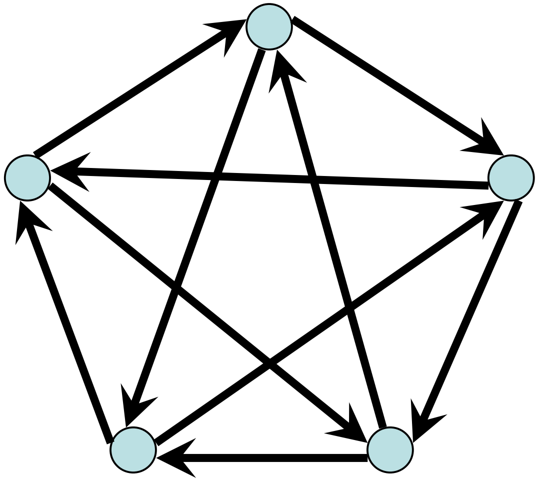


On note $\mathbf{K_n}$ le graphe complet à n sommets.

Types particuliers de graphes (3)

Un graphe complet orienté (antisymétrique) est un **tournoi**.

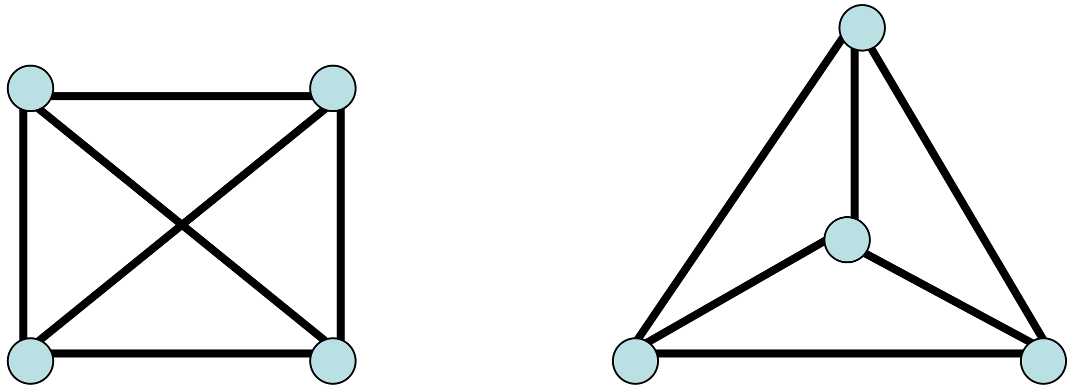




Types particuliers de graphes (4)

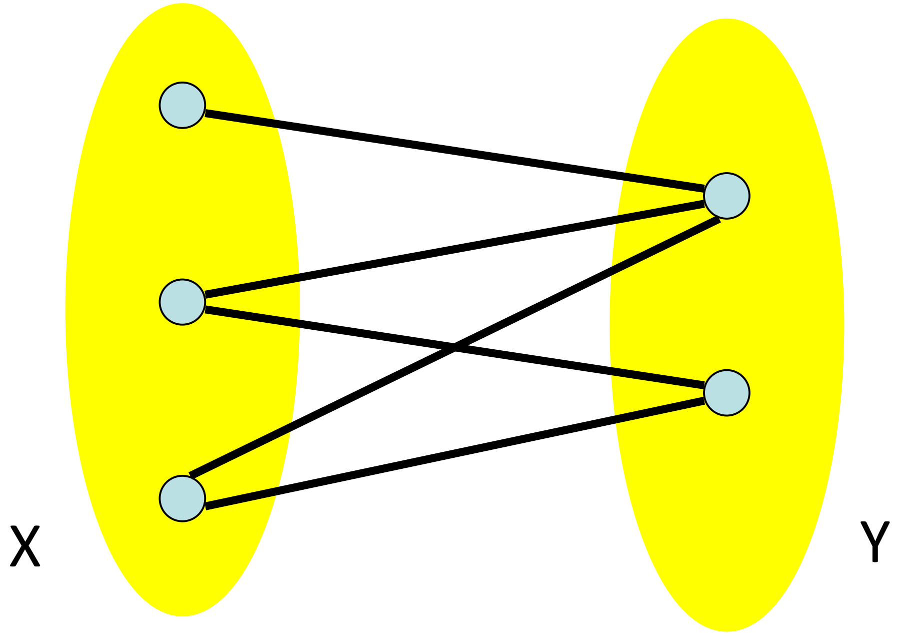
Un graphe est **planaire** s'il peut être dessiné (sur le plan ou la sphère) sans que ses arêtes se croisent.

Exemple :



Types particuliers de graphes (5)

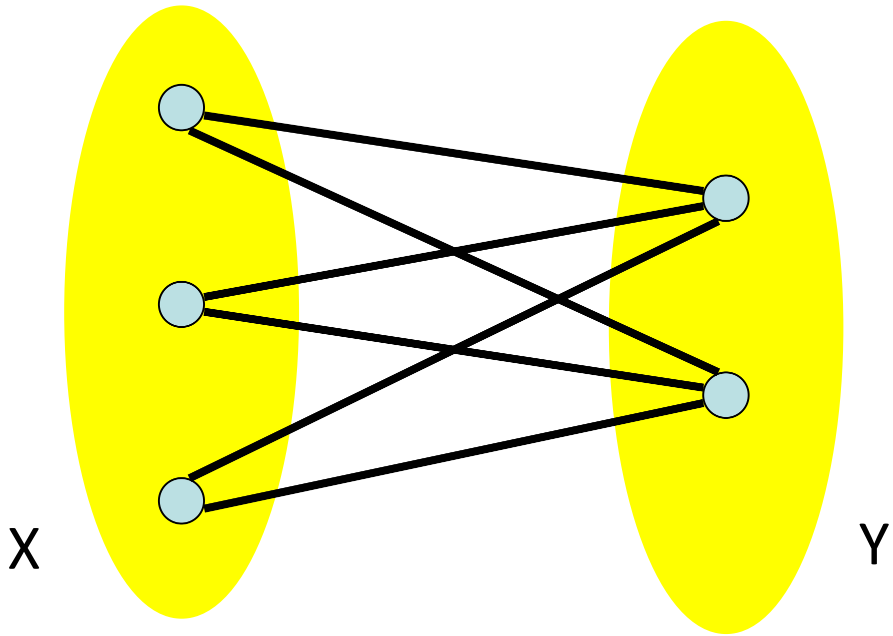
Un graphe est **biparti** s'il existe une partition de l'ensemble de ses sommets, $S = X \cup Y$, telle que toutes les arêtes relient un sommet de X à un sommet de Y.



Types particuliers de graphes (6)

Un graphe est **biparti complet** s'il est biparti et si toutes les arêtes possibles entre X et Y sont présentes.

Le graphe $K_{3,2}$



On note $\mathbf{K_{n,m}}$ le graphe biparti complet avec $|X| = n$ et $|Y| = m$.

Deuxième partie

Les graphes : un outil de modélisation

Modélisation et résolution de problèmes

Problème sur des « objets »

Graphes

Problème de graphes

Solution ?...
(algorithme)

Quelques exemples généraux

Théorie des jeux

- ✓ sommets : positions de jeux,
- ✓ arcs : mouvements de jeux,
- ✓ problème : atteindre une position gagnante

Transport routier (ferroviaire, aérien)

- ✓ graphe : réseau routier (orienté ou non)
- ✓ problèmes : recherche / optimisation de trajet

Réseaux numériques

- ✓ graphe : réseau (orienté ou non)
- ✓ problèmes : routage d'information, optimisation de débit

Etc.

Un problème de plus court chemin (1)

Question :

On souhaite prélever 4 litres de liquide dans un tonneau. Pour cela, on dispose de deux jarres, l'une de 5 litres, l'autre de 3 litres.

Modélisation :

sommets : couples (contenu J5, contenu J3)

arcs : opérations autorisées

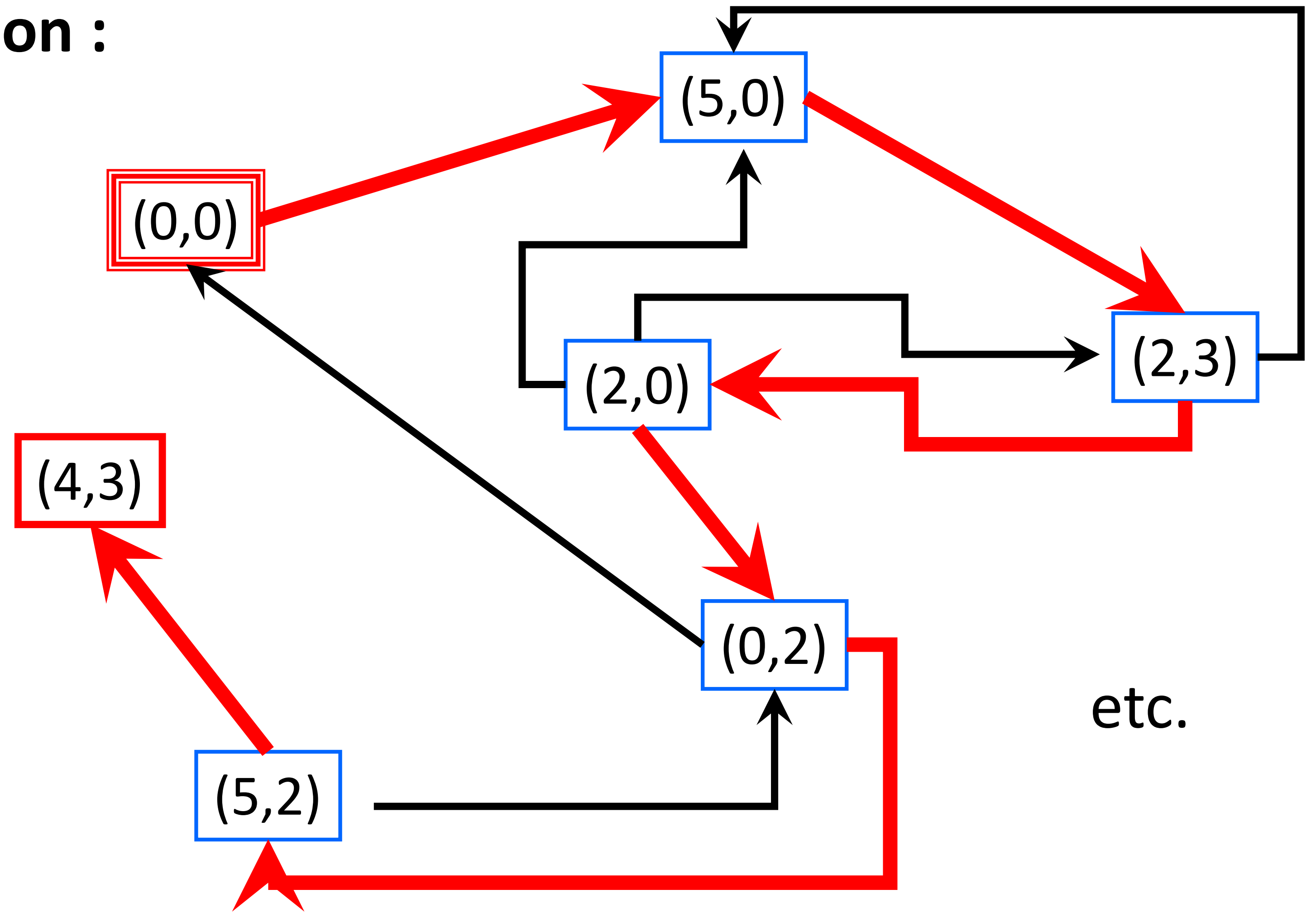
Exemples : (4,1) → (2,3) ou encore (4,1) → (0,1)

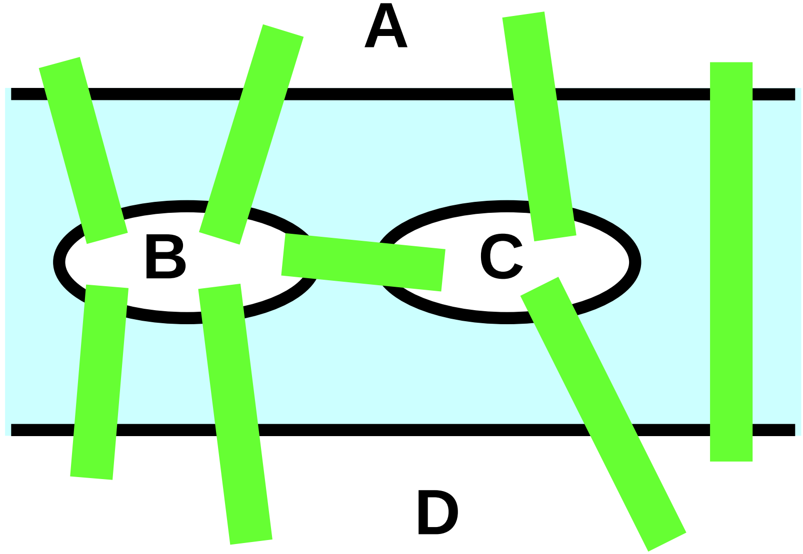
Problème :

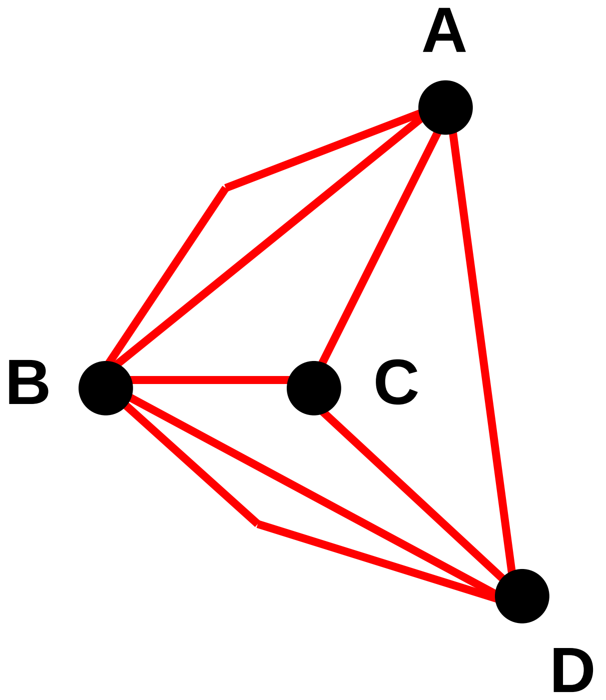
Trouver un plus court chemin reliant (0,0) à (4,x).

Un problème de plus court chemin (2)

Solution :



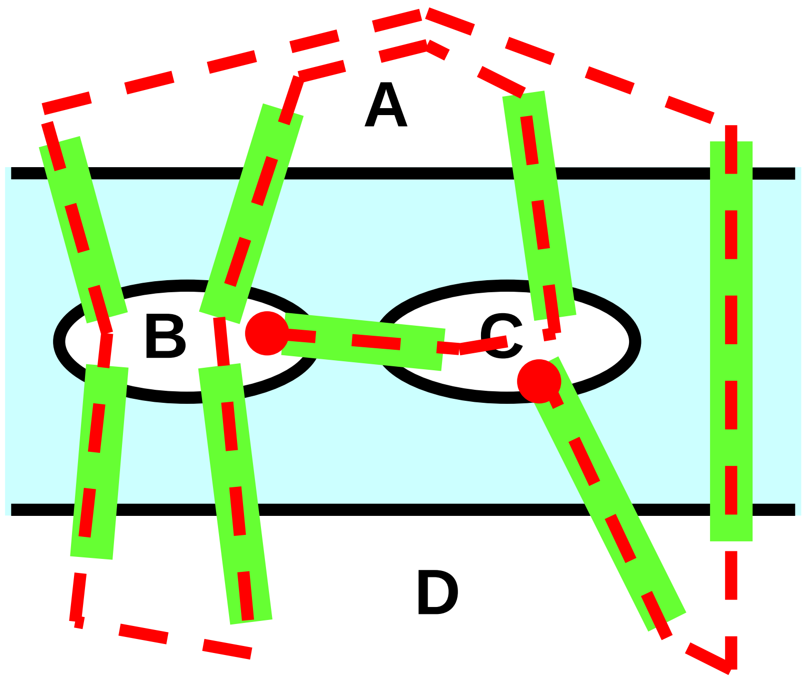


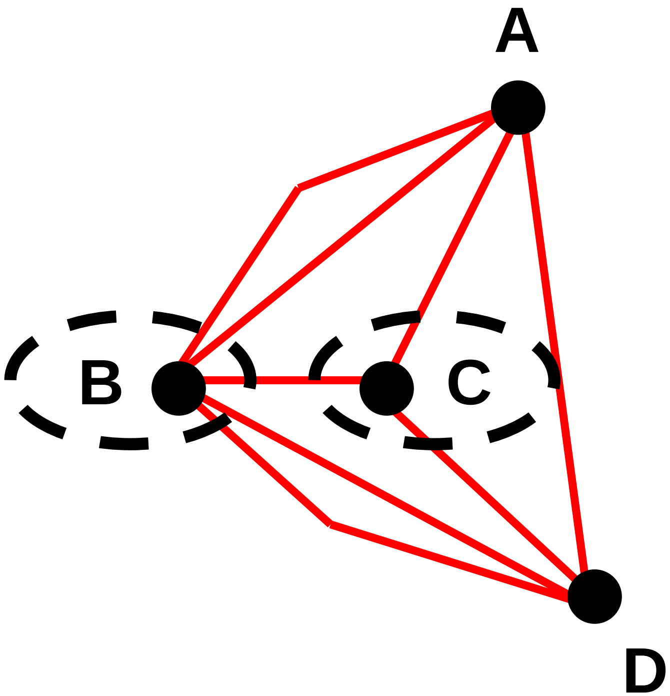


Problème :

Trouver un cycle (chaîne) passant une et une seule fois par chaque arête.

Les ponts de Koenigsberg (2)





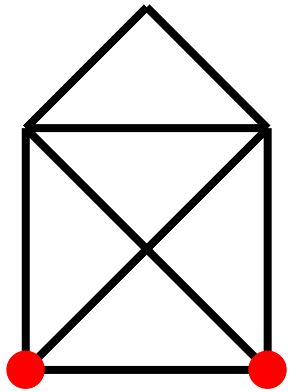
Il existe un **cycle eulérien** si et seulement si G est connexe et tous ses sommets sont de degré pair…

Il existe une **chaîne eulérienne** si et seulement si G est connexe et possède 0 ou 2 sommets de degré impair…

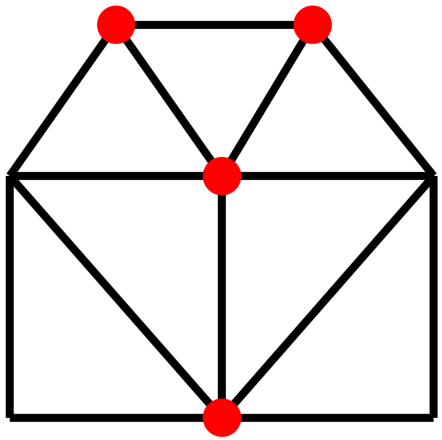
Un problème similaire (petite école)

Problème :

Peut-on tracer les figures suivantes sans lever le crayon ?



OUI



NON

Troisième partie

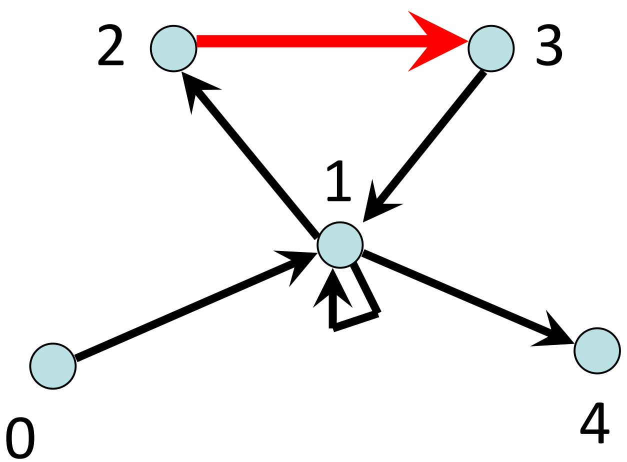
Représentation des graphes (structure de données)

Matrices d'adjacence – Cas orienté

Définition. Soit G = (S,A) un graphe (orienté ou non) avec S = {0, 2, ..., n-1} ; la **matrice d'adjacence** de G, notée M = M(G), est la matrice n x n à valeurs dans {0,1} définie par :

M[i][j] = 1 ssi (i,j) est un arc (arête) de G

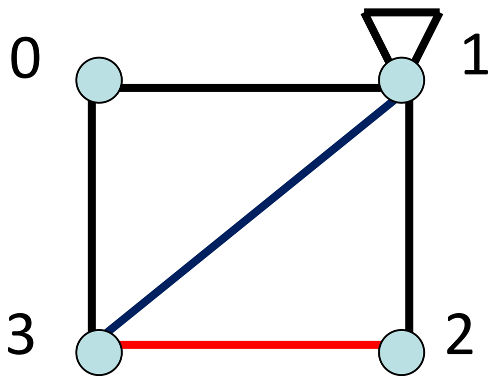
Exemple :



	0	1	2	3	4
0	0	1	0	0	0
1	0	1	1	0	1
2	0	0	0	1	0
3	0	1	0	0	0
4	0	0	0	0	0

Matrices d’adjacence – Cas non orienté

Remarque : dans le cas d’un graphe non orienté, la matrice d’adjacence est **symétrique**.



M :

	0	1	2	3
0	0	1	0	1
1	1	1	1	1
2	0	1	0	**1**
3	1	1	**1**	0

Représentation compacte :

M[i][j] = T[i(i-1)/2 + j] (i > j), ainsi M[3][4]=M[4][3]=T[9]

T :

0	1	1	0	1	0	1	1	**1**	0

1 **2** **3** **4**

Matrices d'adjacence - Complexité

n = nombre de sommets

Complexité en espace : $O(n^2)$

quel que soit le nombre d'arcs, donc coûteux pour les graphes peu denses...

Complexité en temps de quelques opérations :

- tester la présence d'un arc de i vers j : $O(1)$
- parcourir les voisins d'un sommet : $O(n)$
- parcourir tous les arcs : $O(n^2)$

Matrices d'adjacence – Remarque (1)

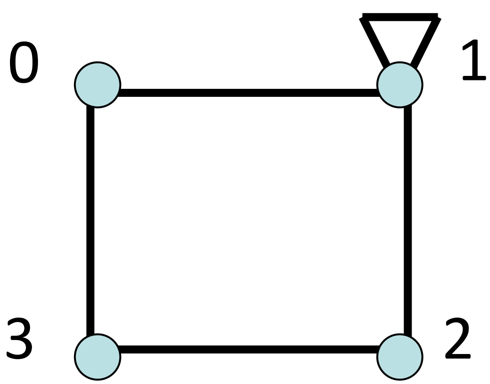
Considérons les opérations (booléennes) suivantes :

+	0	1
0	0	1
1	1	1

x	0	1
0	0	0
1	0	1

Proposition

$M^k[i][j] = 1$ ssi il existe un chemin (chaîne si non orienté) de longueur k allant de i à j



G

	0	1	2	3
0	0	1	0	*1*
1	*1*	*1*	*1*	*0*
2	0	1	0	*1*
3	1	0	1	*0*

M

	0	1	2	3
0	1	1	1	0
1	1	1	1	*1*
2	1	1	1	0
3	0	1	0	1

M^2

Matrices d'adjacence – Remarque (2)

Preuve par récurrence sur k.

1. $k = 1$. $M^1 = M$, donc vrai par définition (arc = longueur 1)

2. Supposons vrai jusqu'au rang k-1.

On a $M^k = M \times M^{k-1}$,
et donc pour tout couple (i,j) :

$$M^k[i,j] = M[i,1] \times M^{k-1}[1,j] + \ldots + M[i,s] \times M^{k-1}[s,j] + \ldots + M[i,n] \times M^{k-1}[n,j]$$

Ainsi, $M^k[i,j] = 1$
ssi il existe s tel que $M[i,s] \times M^{k-1}[s,j] = 1$

i.e. ssi il existe un sommet s tel que
- $M[i,s] = 1$, i.e. l'arc (i,s) existe
- $M^{k-1}[s,j] = 1$, i.e. il existe un chemin de longueur k-1 de s à j.

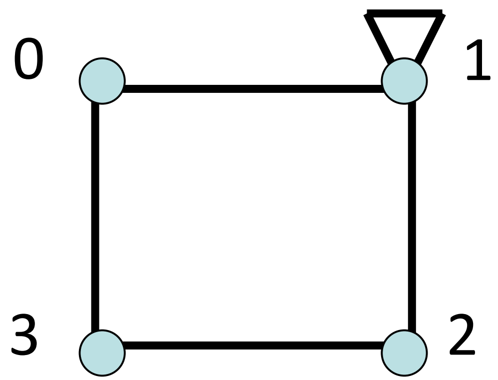
i.e. il existe un chemin de longueur k de i à j.

CQFD.

Matrices d'adjacence – Remarque (3)

En utilisant les opérations + et x usuelles, nous avons :

Proposition. $M^k[i][j] = p$ ssi il existe p chemins (ou chaînes si non orienté) de longueur k allant de i à j



G

	0	**1**	**2**	**3**
0	0	1	0	1
1	1	1	1	0
2	0	1	0	1
3	1	0	1	0

M

	0	**1**	**2**	**3**
0	2	1	2	0
1	1	3	1	2
2	2	1	2	0
3	0	2	0	2

M^2

Preuve : même principe…

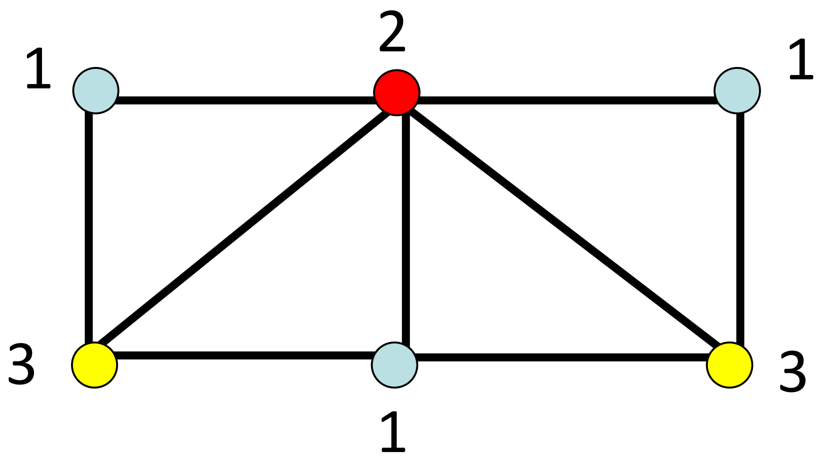
Quatrième partie

Coloration de graphes

k-coloration (1)

Une **k-coloration** (propre) d'un graphe (simple, sans boucles) G est une application c qui associe à chaque sommet de G une couleur de l'ensemble {1, …, k} de façon telle que les sommets voisins ont des couleurs distinctes :

$$\{u, v\} \in E(G) \quad \Rightarrow \quad c(u) \neq c(v)$$

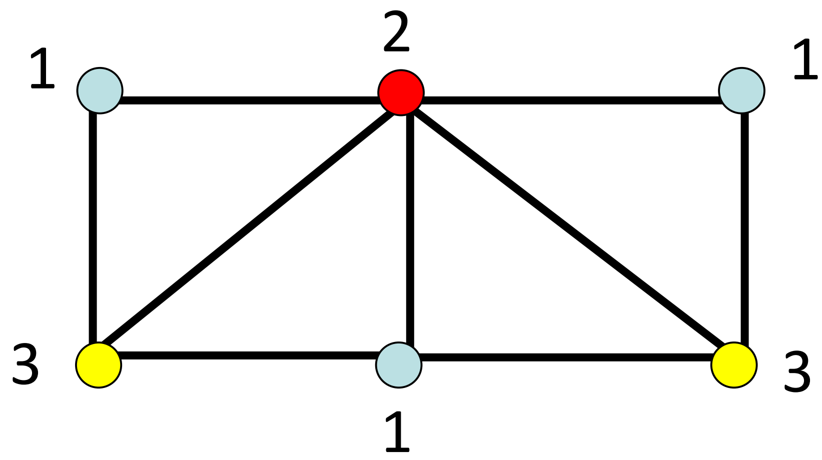


k-coloration (2)

De façon équivalente, une k-coloration de G est une "partition" de l'ensemble V(G) des sommets de G en k ensembles *stables* (ou *indépendants*) :

$$V(G) \;= V_1 \cup \ldots \cup V_k$$

(V_i = ensemble des sommets coloriés i)

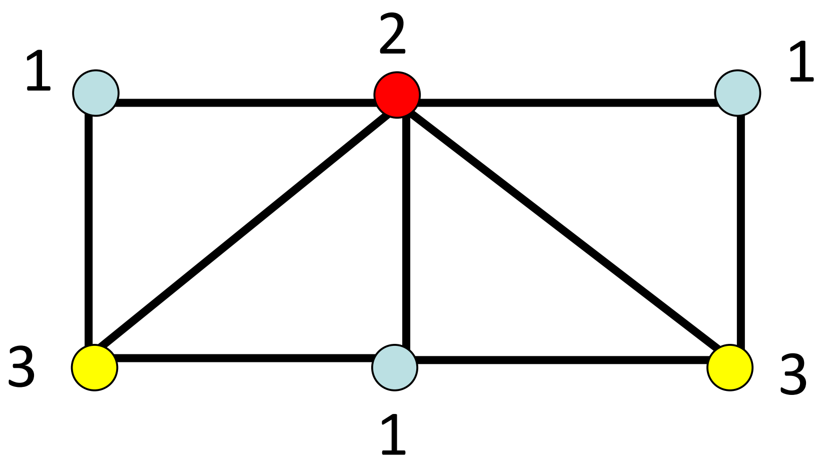


Nombre chromatique

Un graphe G est **k-coloriable** s'il peut être colorié avec k couleurs.

Le **nombre chromatique** d'un graphe G, noté **χ(G)**, est le nombre minimum de couleurs nécessaire pour colorier G.

$\chi(G) = 3$



Un graphe G est **k-chromatique** ssi $\chi(G) = k$.

G k-chromatique ssi G k-coloriable et non (k-1)-coloriable

Algorithme de coloration First-Fit (1)

Un algorithme "simple" de coloration est l'algorithme *glouton* suivant :

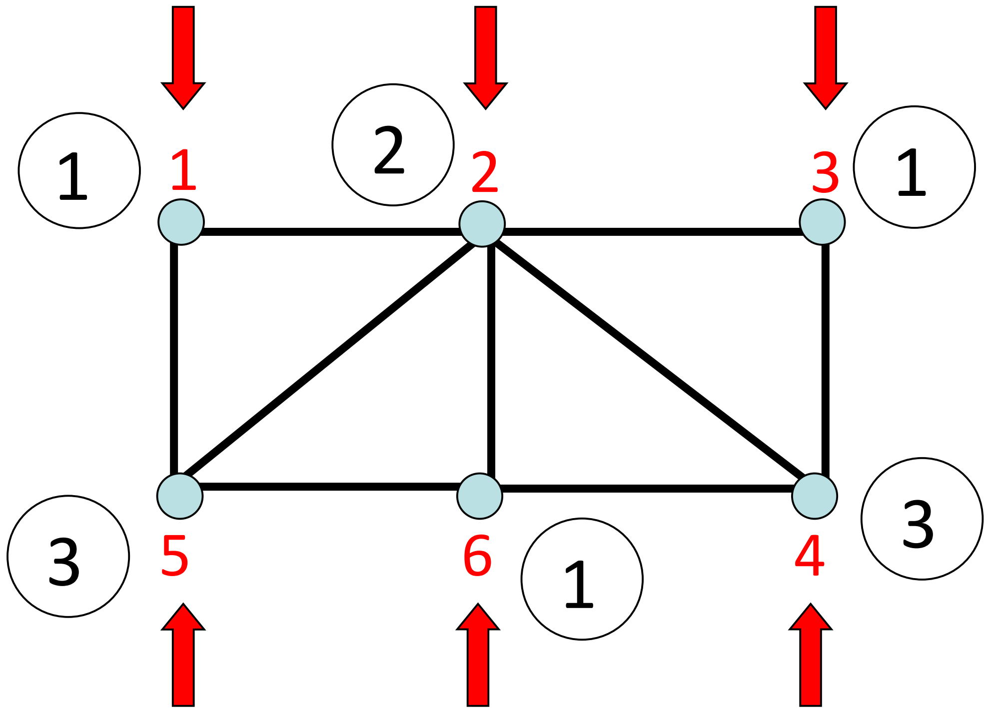
- ordonner les sommets de G : $V(G) = \{ v_1, ..., v_n \}$
- traiter les sommets dans cet ordre, en affectant au sommet v_i la plus petite couleur possible (i.e. distincte des couleurs de ses voisins déjà coloriés)

Remarque. Déterminer si un graphe est k-coloriable est un problème NP-complet pour $k \geq 3$.

Algorithme de Welsh et Powell : ordonner les sommets par degrés décroissants...

Algorithme de coloration First-Fit (2)

Exemple :



Question. Quel est le nombre maximum de couleurs utilisées par cet algorithme ?

Encadrement du nombre chromatique (1)

Proposition. Pour tout graphe G, $\chi(G) \leq \Delta(G) + 1$.

Découle de l'algorithme First-Fit...

Théorème [Brooks, 1941]. Soit G un graphe connexe de degré maximal $\Delta(G)$. Nous avons alors $\chi(G) = \Delta(G) + 1$ si et seulement si

- $\Delta(G) = 2$ et G est un cycle impair ou
- $\Delta(G) \geq 2$ et G est le graphe complet à $\Delta(G) + 1$ sommets.

(Dans le cas contraire, nous avons $\chi(G) <= \Delta(G)$).

Encadrement du nombre chromatique (2)

Proposition. Pour tout graphe G, $\chi(G) \geq \omega(G)$.

$\omega(G)$ = taille maximale d'une clique dans G

(clique = sous-graphe complet)

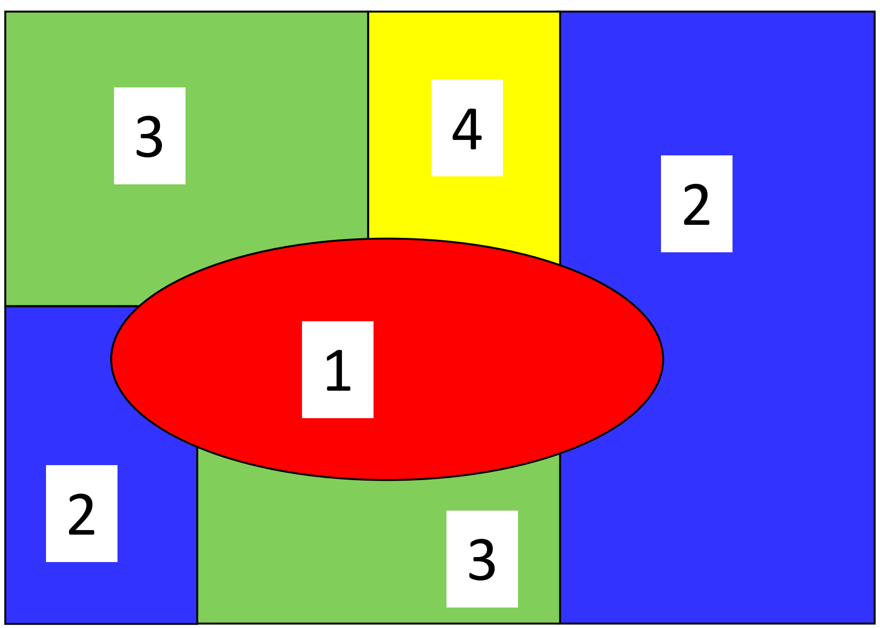
Théorème [Zykov, 1949]. Pour tout k, il existe des graphes k-chromatiques sans triangles.

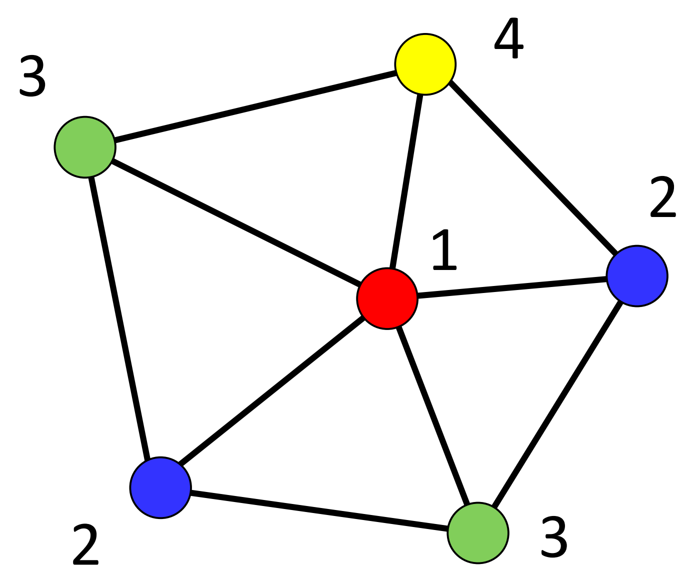
Proposition. Pour tout graphe G, $\chi(G) \geq |V(G)|/\alpha(G)$.

$\alpha(G)$ = taille maximale d'un ensemble stable dans G

Le problème des quatre couleurs (1)

Francis Guthrie (1852). Peut-on colorier toute carte à l'aide de quatre couleurs, de façon telle que les pays ayant une frontière commune aient des couleurs distinctes ?





Le problème des quatre couleurs (2)

Il n'est pas trop difficile de démontrer que les graphes planaires sont 5-coloriables : preuve originale fausse des 4 couleurs due à Kempe (1879), reprise par Heawood (1890)...

Le théorème des quatre couleurs n'a été démontré qu'en 1976, à l'aide d'une preuve utilisant l'ordinateur...

Théorème (Appel et Haken, 1976). Tout graphe planaire est 4-coloriable.

IUT de Bordeaux
Département Informatique

M2201 – Graphes et Langages

Éléments de Théorie des Graphes

Algorithmes de parcours

Principes généraux

De nombreux problèmes nécessitent de parcourir l'ensemble des sommets (ou des arcs, arêtes) d'un graphe.

Ce parcours est nécessairement basé sur la structure du graphe : depuis un sommet, on ne peut atteindre que ses successeurs (cas orienté) ou ses voisins (cas non orienté).

On retrouve deux stratégies classiques :

- parcours en profondeur (depth-first search, DFS)
- parcours en largeur (breadth-first search, BFS)

Dans la suite, on suppose que le graphe est représenté par sa matrice d'adjacence (type TGraphe).

Parcours en profondeur

Parcours en profondeur

Algorithme de Trémaux

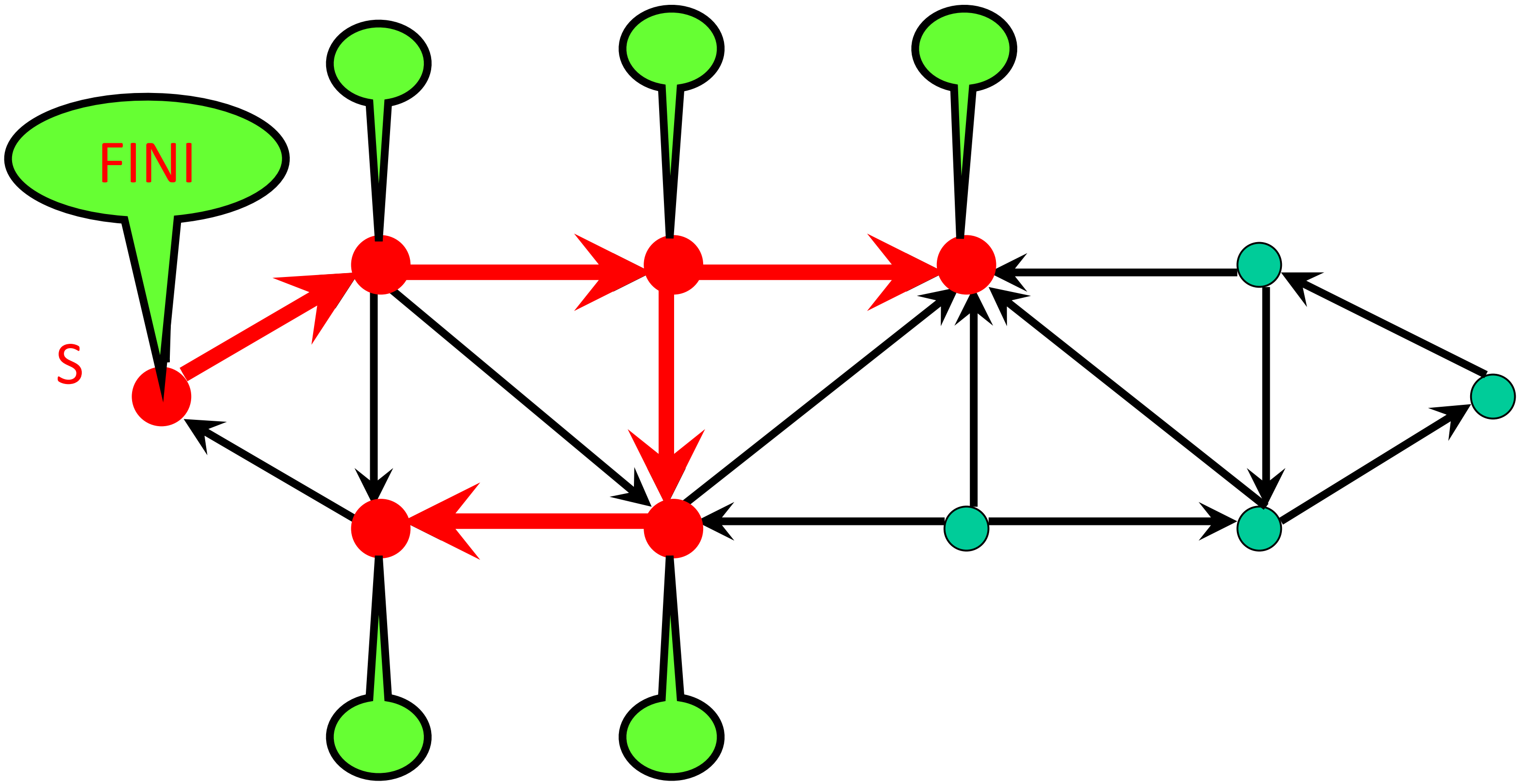
Données.

- un graphe G, un sommet S de G

Principe.

- on marque les sommets parcourus
- à partir d'un sommet, on parcourt récursivement ses successeurs non marqués

Parcours en profondeur - Exemple



Remarque. Seuls les sommets « accessibles » sont atteints !
Il faut « relancer » le parcours...

Algorithme de Trémaux (1)

```
Action DFS ( E G : TGraphe,  E n : entier )

var :     MARQUE : tableau [CMax] de booléens
          S : entier

début
    Pour S de 0 à n – 1 faire
          MARQUE[S] ← faux
    Pour S de 0 à n – 1 faire
          Si ( non MARQUE[S] )
          Alors Trémaux ( G, n, S, MARQUE )
fin
```

Algorithme de Trémaux (2)

```
Action Trémaux ( E G : TGraphe, E n, S : entiers,
          ES MARQUE : tableau [CMax] de booléens)
var :     T : entier
début
   // traiter S si nécessaire ici...
   MARQUE[S] ← vrai
   Pour tout voisin T de S faire
      Si (non MARQUE[T])
      Alors Trémaux (G, n, T, MARQUE)
fin
```

Algorithme de Trémaux - Complexité

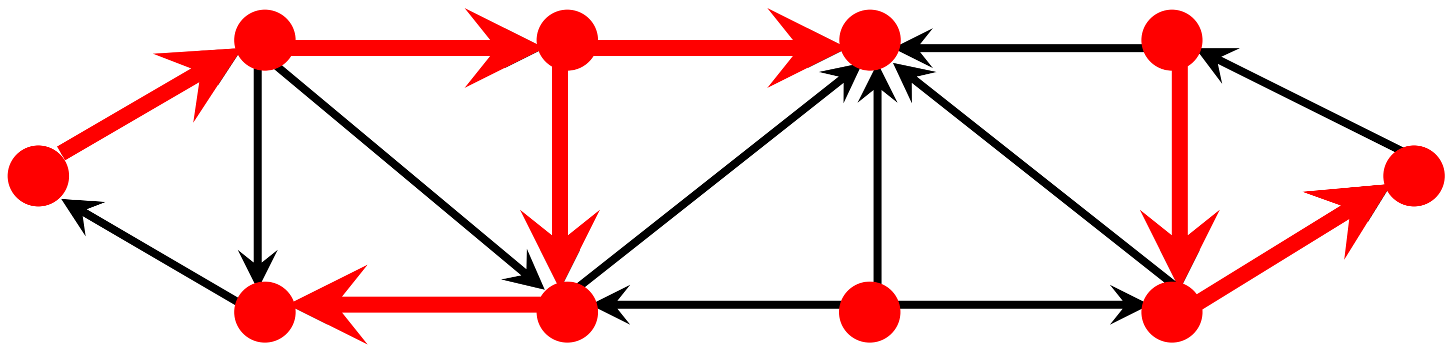
Pour chaque sommet, on doit parcourir la liste de ses successeurs : il faut donc parcourir la ligne correspondante de la matrice d'adjacence.

Chaque sommet n'est traité qu'une seule fois, grâce au tableau MARQUE.

La complexité de cet algorithme est donc en $O(n^2)$.

Exemple (suite)

Fin de l’algorithme sur notre exemple :



Les arcs parcourus (en rouge) constituent une **forêt recouvrante** du graphe G (composée de trois arbres, dont l’un est ici réduit à un sommet).

À quelle condition cette forêt est-elle composée d’un seul arbre (cas orienté, cas non orienté) ?

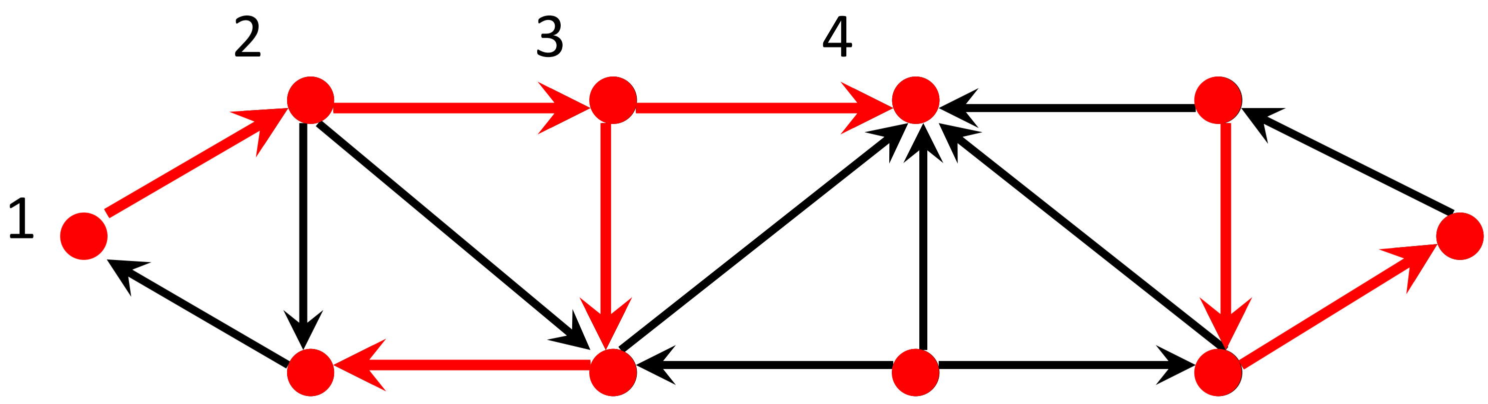
- Nous allons modifier l’algorithme de Trémaux pour construire cette forêt (sans changer sa complexité).

Construction d’une forêt recouvrante (1)

On utilise un tableau PERE qui associe à chaque sommet son père dans la forêt (en prenant par convention PERE[racine] = racine).

Ce tableau PERE peut jouer également le rôle du tableau MARQUE : on l’initialise à -1, un sommet X a alors été marqué ssi PERE [X] $\neq$ -1.

Il permet également d’enlever la récursivité car il contient la « trace » du parcours récursif : lorsque le traitement d’un sommet est terminé, on « remonte » à son père…



Construction d’une forêt recouvrante (2)

```
Action Forêt_Recouvrante
                (   E G : TGraphe,
                    E n : entier,
                    S PERE : tableau [CMax] d'entiers )

var :   S : entier
début
    pour S de 0 à n – 1 faire
        PERE[S] ← -1
    pour S de 0 à n – 1 faire
        Si ( PERE[S] = -1 )
        Alors Arbre_Recouvrant ( G, n, S, PERE )
fin
```

Construction d'une forêt recouvrante (3)

```
Action Arbre_Recouvrant ( E G : TGraphe , E n, S : entiers,
                          ES PERE : tableau [CMax] d'entiers )
var     T , U : entiers
        Fini : booléen
début
    // traiter S si nécessaire ici...
    PERE[S] ← S ; Fini ← Faux ; T ← S
    Tant que ( non Fini ) faire
        Si T possède un successeur U tel que PERE[U] = -1
        Alors PERE[U] ← T ; T ← U          // on descend vers U
        SinonSi  T = S
                     Alors  Fini ← Vrai     // on a terminé
                     Sinon  T ← PERE[T]     // on remonte au père de T
fin
```

Parcours en largeur

Parcours en largeur

Algorithme de parcours en largeur

Données.

- un graphe G, un sommet S de G

Principe.

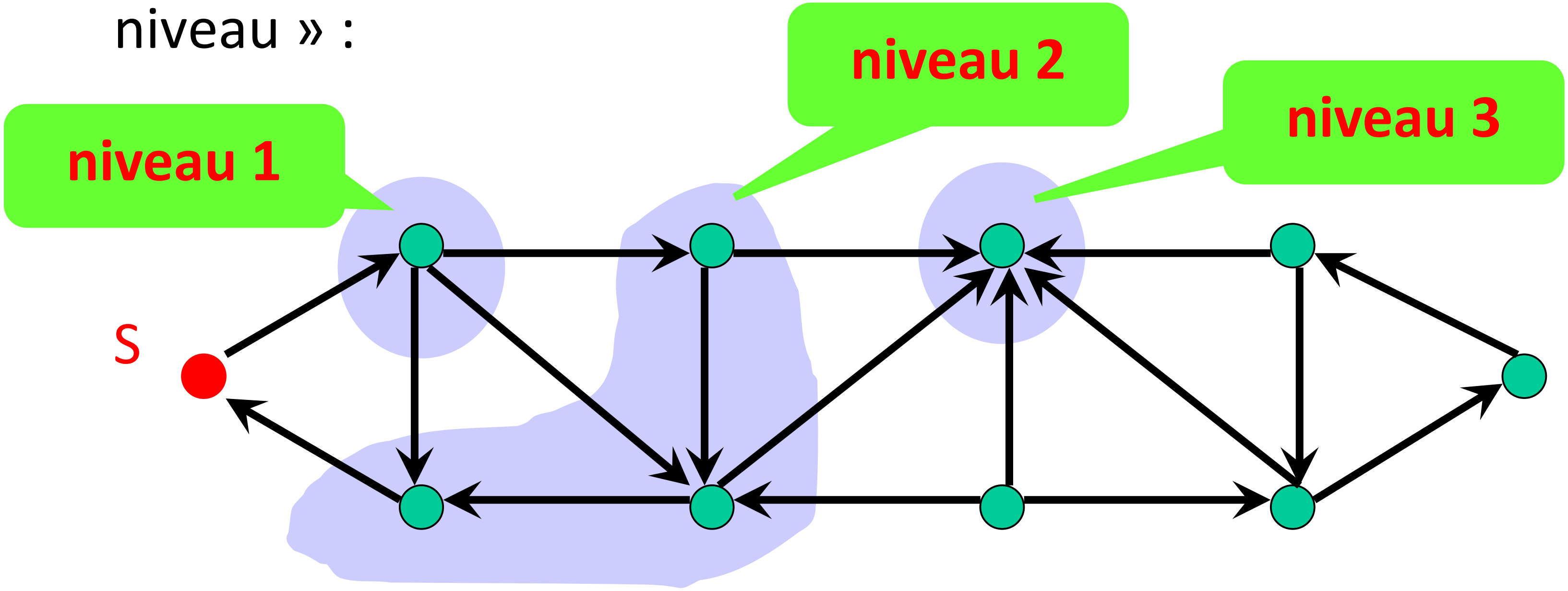
- on marque les sommets parcourus
- à partir d'un sommet, on visite ses successeurs non marqués avant de visiter ses autres descendants

Complexité.

- même complexité que Trémaux : $O(n^2)$
  (seul l'ordre de parcours des sommets change).

Parcours en largeur - Exemple

Cela revient en quelque sorte à parcourir des « courbes de niveau » :



Difficulté : lorsqu'on est arrivé sur le dernier sommet de niveau i, il faut passer aux successeurs du premier sommet de niveau i !...

Solution :

on utilise une <u>file</u> des sommets à traiter.

Parcours en largeur - Algorithme

```
Action Arbre_Recouvrant_Largeur
    ( E G : TGraphe, E n, S : entiers,
        ES PERE : tableau [CMax] d'entiers )

var     T,U : entiers;  F : file d'entiers
début
    CréerFile (F) ; rajouter S à F ; PERE [S] ← S
    Tant que ( non FileVide (F) ) faire
        prendre T dans F
        // traiter T si nécessaire ici...
        Pour tout successeur U de T non marqué faire
            rajouter U à F
            PERE[U] ← T
fin
```

Pour construire une forêt recouvrante, on procède comme précédemment...

Éléments de Théorie des Graphes

Chemins de moindre coût

Quelques définitions / rappels (1)

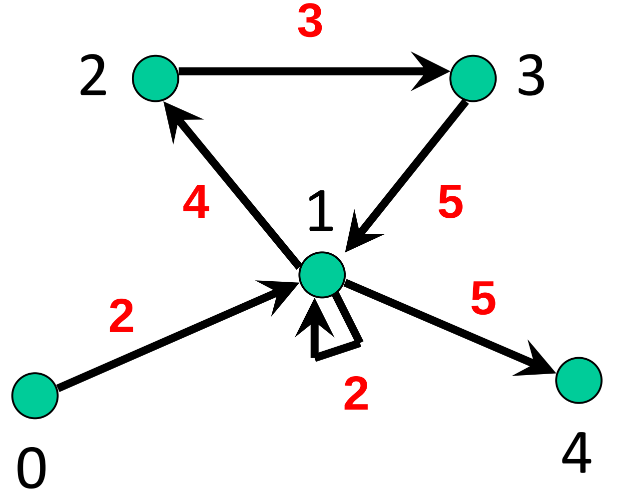
Soit G = (V,E) un graphe orienté (ou non orienté). Un **chemin** (une **chaîne**) **de longueur p** de u à v est une séquence de sommets $(u_0, u_1, \ldots, u_p)$ avec $u_0 = u$, $u_p = v$ et pour tout i, $0 \leq i \leq p-1$, $u_i u_{i+1}$ est un arc (ou une arête) de E.

Un tel chemin est **élémentaire** si tous ses sommets sont distincts, sauf éventuellement le premier et le dernier, auquel cas le chemin est un circuit (ou un cycle).

Quelques définitions / rappels (2)

Un graphe orienté (ou non orienté) **valué** est un triplet $G = (V,E,f)$, ou (V,E) est un graphe et f une fonction associant une valeur, souvent un nombre, à chaque arc (ou arête) de E.

Le **coût** $f(P)$ d'un chemin $P = (u_0, u_1, \dots, u_p)$ dans un graphe valué (V,E,f) est la somme des valeurs des arcs (ou des arêtes) qui le composent.



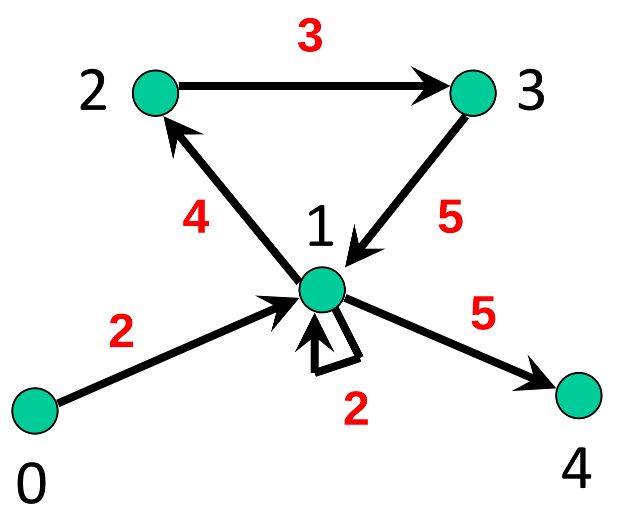
$f(0,1,2,3) = 2 + 4 + 3 = 9$

Remarque. Si $\forall\, e \in E$, $f(e) = 1$, alors $f(P) =$ longueur de P…

Quelques définitions / rappels (3)

Pour représenter un graphe valué, il suffit de stocker la valeur de chaque arc (ou arête) dans la matrice d'adjacence du graphe, **en réservant une valeur particulière**, non utilisée, pour représenter l'**absence** d'arc (ou d'arête).

	0	1	2	3	4
0	0	2	0	0	0
1	0	2	4	0	5
2	0	0	0	3	0
3	0	5	0	0	0
4	0	0	0	0	0



Chemins de moindre coût

Soit G = (V,E,f) un graphe valué, **u** et **v** deux sommets de G.

On va chercher à résoudre les problèmes suivants :

- Trouver un chemin de moindre coût reliant **u** à **v**,
- Trouver des chemins de moindre coût reliant **u** à chacun des autres sommets du graphe,
- Trouver des chemins de moindre coût pour tous les couples de sommets du graphe.

Remarque. On ne peut pas déterminer les chemins de moindre coût si le graphe contient un circuit de coût négatif !...

Algorithme de Floyd

Algorithme de Floyd (1)

L'algorithme de Floyd calcule des chemins de moindre coût pour tous les couples de sommets d'un graphe valué.

Pour cela, il va produire :

- une matrice COUT, de taille n x n, telle que COUT[**u**][**v**] représente le coût d'un chemin de moindre coût reliant **u** à **v**,
- une matrice PRED, de taille n x n, telle que PRED[**u**][**v**] représente le prédécesseur de v dans le chemin de moindre coût reliant **u** à **v**.

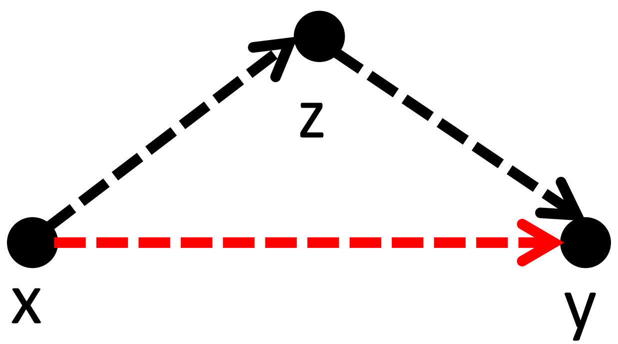
La matrice PRED permet ainsi, de proche en proche, de reconstruire les chemins de moindre coût pour tous les couples de sommets.

Algorithme de Floyd (2)

Initialisations :

- COUT[u][v] = 0 si u = v,
  = f(uv) si uv est un arc,
  = ∞ sinon,
- PRED[u][u] = u pour tout u.

Idée de base de l'algorithme :
amélioration itérative des chemins
par la détection de « raccourcis »…



Si COUT[x][z] + COUT[z][y] < COUT[x][y]

Alors COUT[x][y] ← COUT[x][z] + COUT[z][y]

PRED[x][y] ← PRED[z][y]

Algorithme de Floyd (3)

```
Action FLOYD ( … à compléter … )
var :  x, y, z : entiers
début
    // initialisations
     … à compléter …
    // traitement
    Pour x de 0 à n – 1 Faire
        Pour y de 0 à n – 1 Faire
            Pour z de 0 à n – 1 Faire
                Si COUT[x][z] + COUT[z][y] < COUT[x][y]
                Alors COUT[x][y] ← COUT[x][z] + COUT[z][y]
                      PRED[x][y] ← PRED[z][y]
fin
```

Complexité : clairement en $O(n^3)$…

Algorithme de Bellman-Ford

Algorithme de Bellman-Ford (1)

L'algorithme de Bellman-Ford calcule des chemins de moindre coût reliant un sommet fixé **u** à chacun des autres sommets d'un graphe valué.

Pour cela, il va produire :

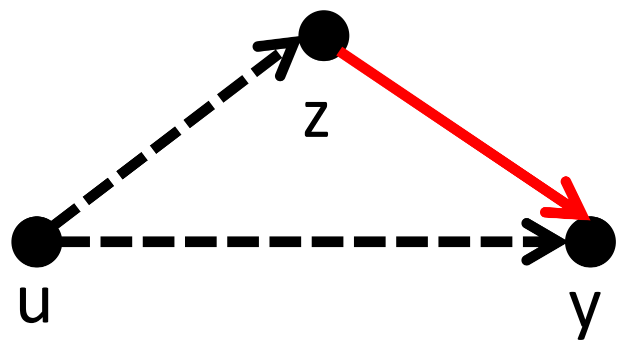
- un tableau COUT, de taille n, tel que COUT[**v**] représente le coût d'un chemin de moindre coût reliant **u** à **v**,
- un tableau PRED, de taille n, tel que PRED[**v**] représente le prédécesseur de v dans le chemin de moindre coût reliant **u** à **v**.

Initialisations :

- COUT[**u**] = 0, COUT[**v**] = ∞ pour tout **v** $\neq$ **u**
- PRED[**v**] = **v** pour tout **v**

Algorithme de Bellman-Ford (2)

L'algorithme de Bellman-Ford repose également sur la détection de raccourcis mais, cette fois, seuls les arcs peuvent constituer des raccourcis…



Si COUT[y] > COUT[z] + f(zy)

Alors COUT[y] ← COUT[[z] + f(zy)

PRED[y] ← z

Il va falloir (re-)parcourir les arcs tant que l'on trouve des raccourcis (car tout nouveau raccourci peut améliorer des chemins déjà traités…).

Algorithme de Bellman-Ford (3)

```
Action BELLMAN-FORD ( … à compléter … )
var :  y, z : entiers ;  fini : booléen
début
    // initialisations
     … à compléter …  ; fini ← faux
    // traitement
    Tant Que non fini Faire
        fini ← vrai
        Pour tout arc zy de G Faire
            Si COUT[y] > COUT[z] + f(yz)
            Alors    COUT[y]  ←  COUT[z] + f(zy)
                     PRED[y]  ←  z  ;  fini  ←  faux
fin
```

Complexité : toujours en $O(n^3)$…

Algorithme de Dijkstra

Algorithme de Dijkstra (1)

L'algorithme de Dijkstra calcule également des chemins de moindre coût reliant un sommet fixé **u** à chacun des autres sommets d'un graphe valué.

Comme l'algorithme précédent, il va produire deux tableaux de taille n, COUT et PRED.

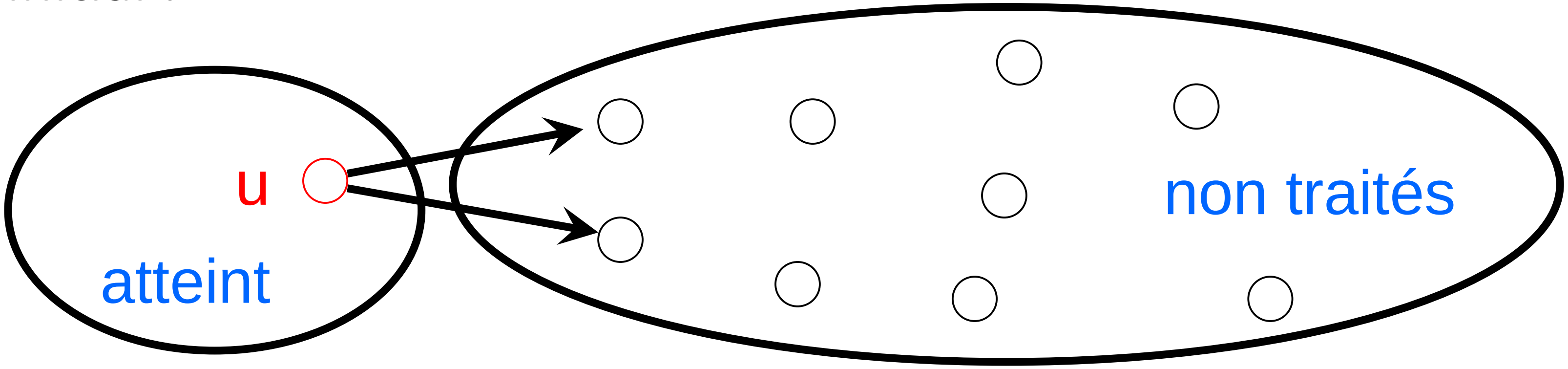
L'algorithme de Dijkstra a une complexité en $O(n^2)$ mais **il n'est utilisable que si toutes les valeurs des arcs sont positives...**

On va associer à chaque sommet **x** un ***état*** pouvant prendre trois valeurs :

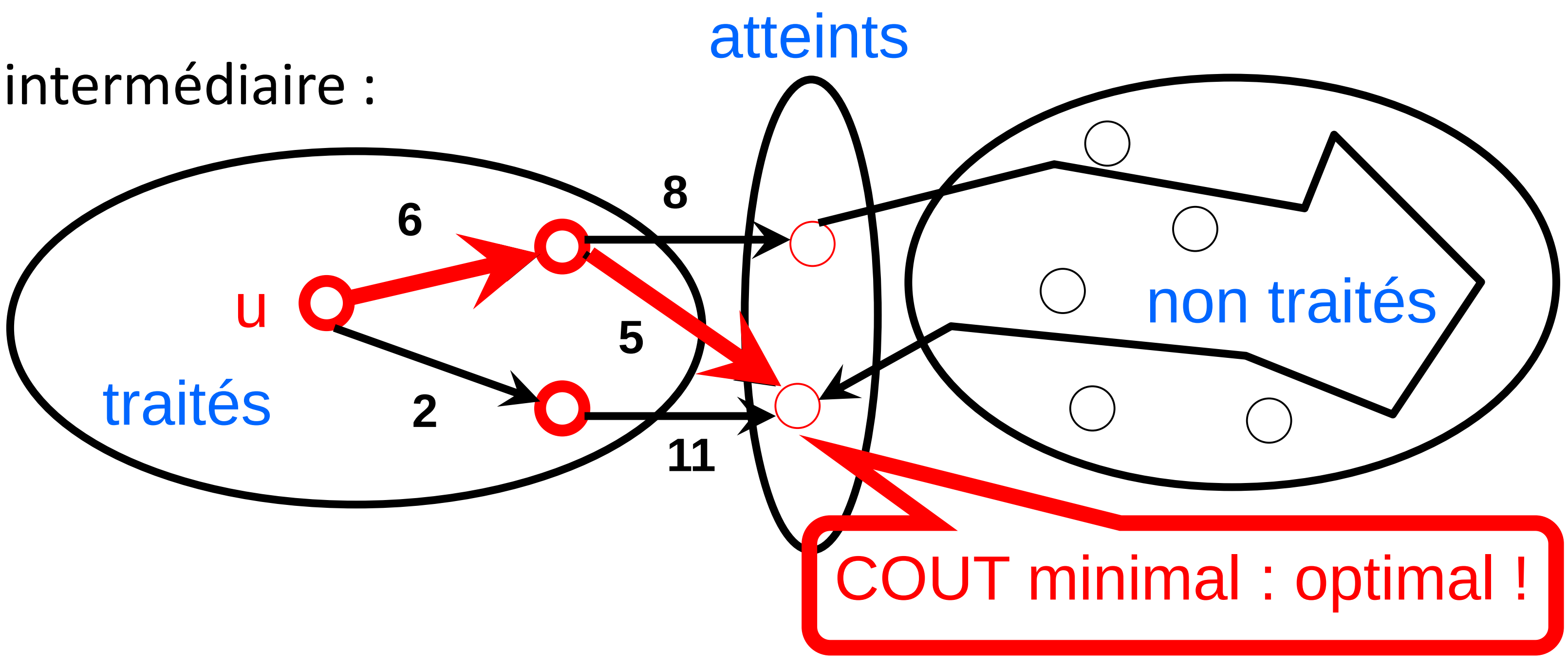
- **traité** : on connaît un chemin <u>optimal</u> de **u** à **x**
- **atteint** : on connaît un chemin de **u** à **x**
- **non traité** : on ne connaît aucun chemin de **u** à **x**

Algorithme de Dijkstra (2)

État initial :



État intermédiaire :



Algorithme de Dijkstra (3)

Initialisations :

- le sommet **u** est dans l'état **atteint**, les autres dans l'état **non traité**.
- COUT et PRED : comme précédemment…

Principe de l'algorithme :

Soit **x** le sommet dans l'état **atteint** dont l'attribut COUT est minimal parmi tous les sommets dans l'état **atteint**.

1. **x** passe dans l'état **traité**,
2. On met à jour les attributs COUT et PRED des successeurs de **x** qui ne sont pas dans l'état **traité** et ils passent dans l'état **atteint**.
3. L'algorithme s'arrête lorsque plus aucun sommet n'est dans l'état **atteint** (tous les sommets atteignables depuis **u** sont dans l'état **traité**).

Éléments de Théorie des Graphes

Arbres couvrants de coût minimal

Arbre couvrant de coût minimal (1)

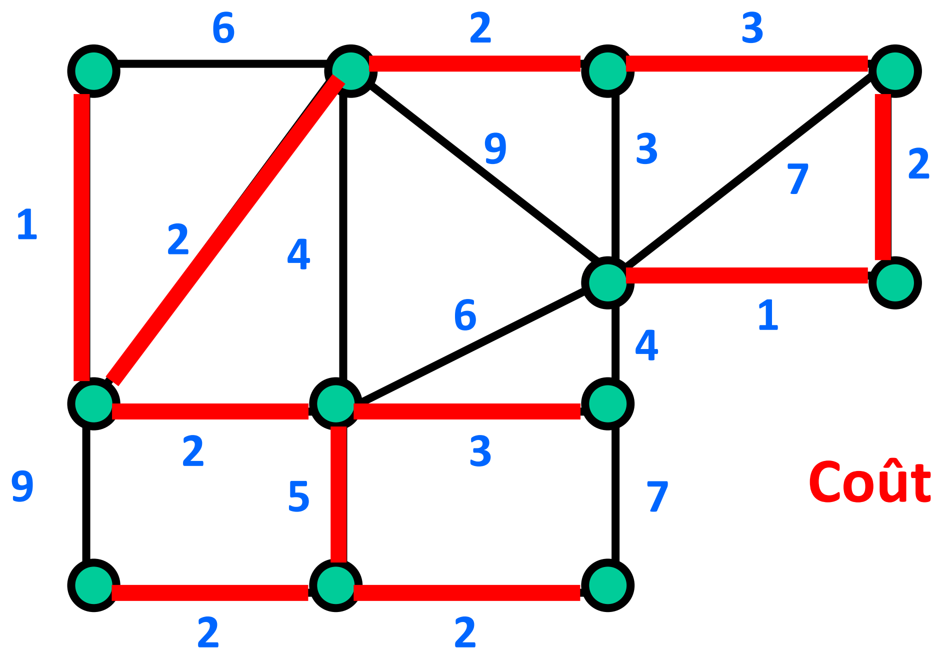
On considère un graphe non orienté **connexe et valué**, G = (S, A, f), avec f une fonction associant à chaque arête une valeur entière (coût).

Un **arbre couvrant** (ou AC) de G est un arbre composé d'arêtes de G et contenant tous les sommets de G.

Le **coût d'un AC** est alors la somme des coûts des arêtes qui le composent.

On cherche ici à construire un arbre couvrant de coût minimal, ou **ACM**.

Arbre couvrant de coût minimal (2)



Coût = 25

Question. Un graphe admet-il un unique ACM ?

Algorithme de Prim

Algorithme de Prim : principe

Donnée : un graphe non orienté connexe valué G,

Résultat : un ensemble d'arêtes (composant un ACM).

Principe.

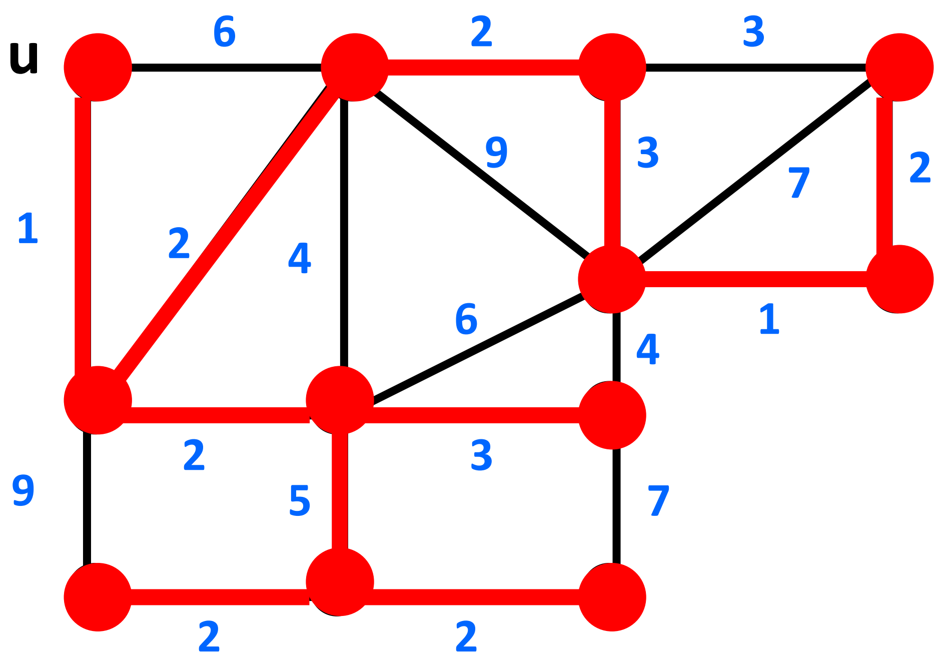
- ✓ On choisit un sommet quelconque **u** que l'on place dans un ensemble **S**.
- ✓ À chaque étape, on choisit une arête **vw** de coût minimal et « sortant » de **S** (i.e. **v** $\in$ **S** et **w** $\notin$ **S**). On rajoute cette arête à l'ACM et le sommet **w** à **S**.
- ✓ On s'arrête lorsqu'on a choisi n – 1 arêtes…

- ➢ C'est un algorithme **glouton** : on ne revient pas sur une décision (une arête choisie sera dans l'ACM).
- ➢ À tout instant, le sous-graphe induit par les arêtes de S est **connexe** et **sans cycle** (arbre).

Algorithme de Prim : exemple

Exemple d'exécution.

- **S** = sommets rouges
- arêtes choisies en rouge



Coût = 0

Coût = 1

Coût = 3

Coût = 5

Coût = 7

Coût = 10

Coût = 11

Coût = 13

Coût = 16

Coût = 21

Coût = 23

Coût = 25

Algorithme de Prim

S : ensemble de sommets
ACM : ensemble d'arêtes

…
ACM = $\emptyset$
Choisir un sommet **u**
S $\leftarrow$ { **u** }
Pour i de 1 à n – 1 Faire
    Choisir une arête **vw** de coût minimum
        avec **v** $\in$ **S** et **w** $\notin$ **S**
    ACM $\leftarrow$ **ACM** $\cup$ { **vw** }
    S $\leftarrow$ **S** $\cup$ { **w** }

Algorithme de Prim : questions…

- Comment représenter l'ensemble **S** ?
- Comment réaliser l'opération **S** $\leftarrow$ **S** $\cup$ { **w** } ?
- Comment représenter l'ensemble **ACM** ? son cardinal ?
- Comment réaliser l'opération **ACM** $\leftarrow$ **ACM** $\cup$ { **vw** } ?
- Comment choisir une arête **vw** de coût minimum avec **v** $\in$ **S** et **w** $\notin$ **S** ?
- Quelle est la complexité de ces opérations ? de l'algorithme ?

Il n'y a donc plus qu'à écrire l'algorithme !…

Algorithme de Kruskal

Algorithme de Kruskal : principe

Donnée : un graphe non orienté connexe valué G,

Résultat : un ensemble d'arêtes (composant un ACM).

Principe.

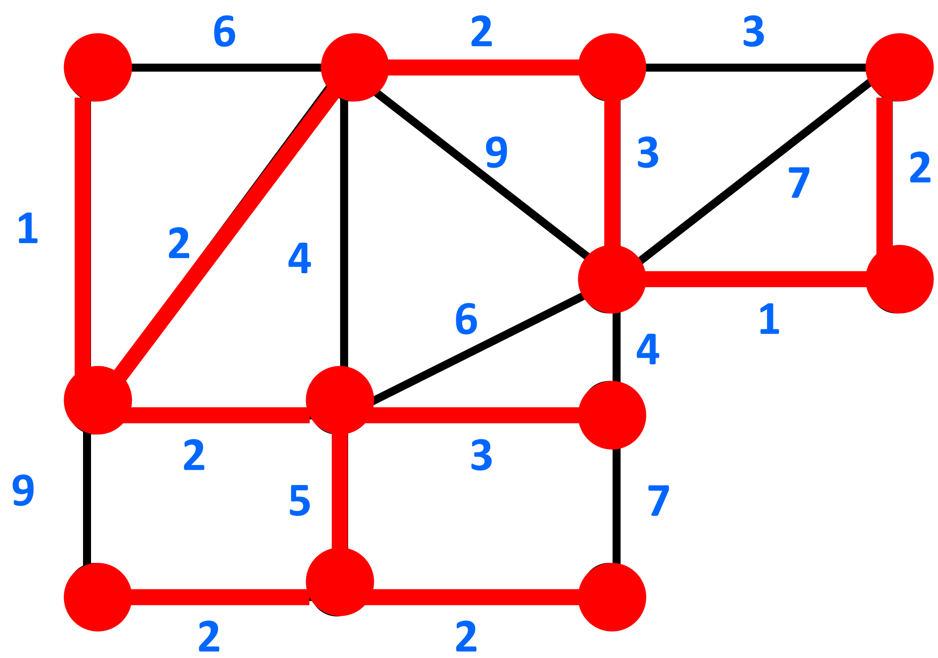
- ✓ On trie au préalable les arêtes par coûts croissants.
- ✓ On part d'une forêt couvrante sans aucune arête, i.e. chaque sommet est un arbre...
- ✓ À chaque étape, on choisit une arête **vw** de coût minimum joignant deux arbres distincts de la forêt. On rajoute cette arête à l'ACM.
- ✓ On s'arrête lorsqu'il n'y a plus qu'un seul arbre.

> - C'est encore un algorithme glouton...
> - À tout instant, le sous-graphe induit par les arêtes de S est **sans cycle** (forêt).

Algorithme de Kruskal : exemple

Exemple d'exécution.

- arêtes choisies en rouge



Coût = 0

Coût = 1

Coût = 2

Coût = 4

Coût = 6

Coût = 8

Coût = 10

Coût = 12

Coût = 14

Coût = 17

Coût = 20

Coût = 25

Algorithme de Kruskal

L, ACM : ensemble d'arêtes
ACM : ensemble d'arêtes
i : entier

…
ACM $\leftarrow \varnothing$
Trier les arêtes de G par coûts croissants
et les ranger dans L
Tant Que | ACM | < n – 1 Faire
retirer la première arête **vw** de L
Si (ACM $\cup$ { **vw** } est sans cycle)
Alors ACM $\leftarrow$ ACM $\cup$ { **vw** }

Algorithme de Kruskal : question…

Comment représenter ACM (forêt couvrante) ?

- un tableau d'entiers (tableau PÈRE)
- seule difficulté : tester si ACM ∪ { **vw** } est sans cycle...

Tableau PÈRE : chaque arbre de la forêt possède une racine.

vw crée un cycle
ssi Rac(**v**) = Rac(**w**).

Ensuite, on renverse dans ACM (tableau PÈRE) le chemin de Rac(**v**) vers **v**... et PÈRE[**v**] ← **w**.